Nano-Quiz Makeups

Wednesday, May 4, 6-11pm, 34-501.

- everyone can makeup/retake NQ 1
- everyone can makeup/retake two additional NQs
- you can makeup/retake other NQs excused by S³

If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower!

Design Lab 12: One-Dimensional Localizer

As robot drives along hallway with obstacles to its side, estimate its current position based on previous estimates and sonar information.

State $S_t$: discretized values of distance along the hallway ($x$).

Transition model $Pr(S_{t+1} = s' | S_t = s)$: conditional distribution of next state given current state.

Observation model $Pr(O_t = d | S_t = s)$: conditional distribution of left-facing sonar readings ($y$) given state.

Planning

Make a plan by searching.

Example: Eight Puzzle

Rearrange board by sequentially sliding tiles into the free spot.
Check Yourself

How many different board configurations (states) exist?

Start

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Goal

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

1. $8^2 = 64$
2. $9^2 = 81$
3. $8! = 40320$
4. $9! = 362880$
5. none of the above

Search Algorithm

Develop an algorithm to systematically conduct a search.

Analyze how well the algorithm performs.

Optimize the algorithm:
- find the “best” solution (i.e., minimum path length)
- by considering as few cases as possible.

Algorithm Overview

Find minimum distance path between 2 points on a rectangular grid.

Represent all possible paths with a tree (shown to just length 3).

Find the shortest path from A to I.

Algorithm Overview

The tree could be infinite.

Therefore, we will construct the tree and search at the same time.

Python Representation

Represent possible locations by states: 'A','B','C','D',...'I'

Represent possible transitions with successor procedure
- inputs: current state (location) and action (e.g., up, right, ...)
- output: new state

Define initialState (starting location)

Determine if goal has been achieved with goalTest procedure
- input: state
- output: True if state achieves goal, False otherwise.

Python Representation


actions = [0, 1, 2, 3]

def successor(s,a):
    if a<len(successors[s]): return successors[s][a] else: return s

def goalTest():
    def initialState = 'A'
    initialState = 'A'
Search Trees in Python

Represent each node in the tree as an instance of class `SearchNode`.

```
class SearchNode:
    def __init__(self, action, state, parent):
        self.action = action
        self.state = state
        self.parent = parent
    def path(self):
        if self.parent is None:
            return [(self.action, self.state)]
        else:
            return self.parent.path() + [(self.action, self.state)]
```

Search Algorithm

Construct the tree and find the shortest path to the goal.

```
Algorithm:
- initialize agenda (list of nodes being considered)
  to contain starting node
- repeat the following steps:
  - remove one node from the agenda
  - add that node's children to the agenda
  until goal is found or agenda is empty
- return resulting path
```

Search Algorithm in Python

Repeatedly (1) remove node (parent) from agenda and (2) add parent's children until goal is reached or agenda is empty.

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState)]
    while not empty(agenda):
        parent = getElement(agenda)
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                add(newN, agenda)
        return None
```

Order Matters

Replace first node in agenda by its children:

```
step Agenda
0:  A
1:  AB AD
2:  ABA ABC ABE AD
3:  ABAB ABAD ABC ABF ABF AD

Depth First Search
```

Order Matters

Replace last node in agenda by its children:

```
step Agenda
0:  A
1:  AB AD
2:  ABADA ADE ADG

also Depth First Search
```

Order Matters

Remove first node from agenda. Add its children to end of agenda.

```
step Agenda
0:  ADG ABAB ABAD ABC BCF ABF ABF
    ABF ABEH ADAB ADAD ADEB ADED ADEF ADEH
1:  ABAB ABAD ABC BCF ABF ABF ABF
    ADAB ADAD ADEB ADED ADEF ADEH ADGD ADGH

Breadth First Search
```
Order Matters

Replace last node by its children (depth-first search):
- implement with stack (last-in, first-out).

Remove first node from agenda. Add its children to the end of the agenda (breadth-first search):
- implement with queue (first-in, first-out).

Stack

Last in, first out.

```python
def search(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                agenda.push(newN)
    return None
```

Stack Class

Last in, first out.

class Stack:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop()
    def empty(self):
        return self.data is []

```python
>>> s = Stack()
>>> s.push(1)
>>> s.push(9)
>>> s.push(3)
>>> s.pop()
3
>>> s.pop()
9
>>> s.push(-2)
>>> s.pop()
-2
```

Queue

First in, first out.

```python
def search(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                agenda.push(newN)
    return None
```

Queue Class

First in, first out.

class Queue:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop(0)  # NOTE: different argument
    def empty(self):
        return self.data is []

```python
>>> q = Queue()
>>> q.push(1)
>>> q.push(9)
>>> q.push(3)
>>> q.pop()
1
>>> q.pop()
9
>>> q.push(-2)
>>> q.pop()
3
```

Depth-First Search

Replace getElement, add, and empty with stack commands.

```python
def search(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                agenda.push(newN)
    return None
```
Breadth-First Search
Replace `getElement`, `add`, and `empty` with `queue` commands.

```python
def search(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                agenda.push(newN)
    return None
```

Too Much Searching
Find minimum distance path between 2 points on a rectangular grid.

```
A B C
D E F
G H I
```

Represent all possible paths with a tree (shown to just length 3).

Pruning
Prune the tree to reduce the amount of work.

Pruning Rule 1:
Don’t consider any path that visits the same state twice.

```
A C E
D F H
```

Pruning Rule 1
Implementation (depth first, switch to `Queue` for breadth first)

```python
def search(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif parent.inPath(newS):  # pruning rule 1
                pass
            else:
                agenda.push(newN)
    return None
```
Pruning Rule 1

Add `inPath` to `SearchNode`.

class SearchNode:
    def __init__(self, action, state, parent):
        self.action = action
        self.state = state
        self.parent = parent

    def path(self):
        if self.parent == None:
            return [(self.action, self.state)]
        else:
            return self.parent.path() + [(self.action, self.state)]

    def inPath(self, state):
        if self.state == state:
            return True
        elif self.parent == None:
            return False
        else:
            return self.parent.inPath(state)

Pruning Rule 2

If multiple actions lead to the same state, consider only one of them.

Depth-First Search Example

```python
def search(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                # pruning rule 2
                pass
            elif parent.inPath(newS):
                # pruning rule 1
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

Breadth-First Search

```python
def search(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                # pruning rule 2
                pass
            elif parent.inPath(newS):
                # pruning rule 1
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

Depth-First Search Properties

- May run forever if we don’t apply pruning rule 1.
- May run forever in an infinite domain.
- Doesn’t necessarily find the shortest path.
- Efficient in the amount of space it requires to store the agenda.
Breadth-First Search Example

Breadth-First Search Properties

- Always returns a shortest path to a goal state, if a goal state exists in the set of states reachable from the start state.
- May run forever in an infinite domain if there is no solution.
- Requires more space than depth-first search.

Still Too Much Searching

Breadth-first search, visited 16 nodes: but there are only 9 states!

Dynamic Programming Principle

The shortest path from $X$ to $Z$ that goes through $Y$ is made up of
- the shortest path from $X$ to $Y$ and
- the shortest path from $Y$ to $Z$.

We only need to remember the shortest path from the start state to each other state!

Dynamic Programming in Breadth-First Search

The first path that BFS finds from start to $X$ is the shortest path from start to $X$.

We only need to remember the first path we find from the start state to each other state.

Dynamic Programming as a Pruning Technique

Don't consider any path that visits a state that you have already visited via some other path.

Need to remember the first path we find to each state.

Use dictionary called visited
Breadth-First Search with Dynamic Programming

```python
def breadthFirstDP(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    visited = {initialState: True}
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif visited.has_key(newS):  # rules 1, 2, 3
                pass
            else:
                visited[newS] = True
                agenda.push(newN)
    return None
```

Breadth-First with Dynamic Programming Example

```
A — B — C
|
D — E — F
|
G — H — I
```

Summary

Developed two search algorithms
- depth-first search
- breadth-first search

Developed three pruning rules
- don’t consider any path that visits the same state twice
- if multiple actions lead to same state, only consider one of them
- dynamic programming: only consider the first path to a given state

- everyone can makeup/retake NQ 1
- everyone can makeup/retake two additional NQs
- you can makeup/retake other NQs excused by S^3

If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower!